Deep Memory Profiler in Chromium

Author: Dai Mikurube (dmikurube@chromium.org)
Last updated on: Sep 25, 2012

Objective

This document describes an approach of the Deep Memory Profiler in Chromium. It doesn't describe the architecture and the code.

It profiles memory usage not only in a V8 JavaScript heap, but in an entire Chromium process memory. The profiler is :

- to find suspicious (blamed) components of memory bloat
- to track memory usage break-down continuously for Chromium versions
- NOT to find causes of memory bloat in JavaScript
    - The V8 JavaScript heap profiler is better for the purpose.
- NOT to find reproducing conditions of memory bloat
    - The profiler works for reproduced issues.

It targets engineers familiar with Chromium internals at first. Then, it extends its target to web application developers using Chromium.

We plan to integrate this profiler to the long-running performance testing framework, endurance tests.

Background

Memory bloat has been a serious issue in Chromium for years. Bloat is harder to fix than leak and errors. We have memory checkers like Valgrind and Address Sanitizer for memory leak and errors, but no handy tools for bloat.

Continuous perf tests have been the only signal to find memory bloat. But, 1) each test runs for very short time, and 2) they report only total memory usage. Memory bloat doesn't become evident for (1). We have to use overkilling memory checkers and profilers and spend much time for (2). DMP is an easy-to-use tool for bloat to know "who is the memory eater".

Tackling memory issues requires experienced techniques. It makes experienced engineers overloaded. Efforts to reduce memory usage don't catch up with increasing memory usage. We already have an awesome heap profiler for V8, but it focuses on JavaScript heap. It doesn't understand C++ objects. We also have a C/C++-level heap profiler in TCMalloc. But, it needs experts, and it's not good for automatic testing and profiling.

It's easy to instrument to trace memory usage in some suspicious classes, for example, in

constructors of DOM objects. But, we cannot cover all allocation. It's hard to blame "untraced" allocation.

The Deep Memory Profiler (DMP) is expected to work as "the first profiler" to narrow down the cause of memory bloat by looking at an entire process memory.

Design

To look at the entire process memory, DMP hooks mmap and malloc calls in Chromium. Available information is stack traces for mmap and malloc. It requires experienced skills and deep knowledge in Chromium to analyze stack traces manually.

Classifying the stack traces by pattern matching is a good way for rough understanding. DMP does this. Post-mortem is acceptable since we have many stack traces.

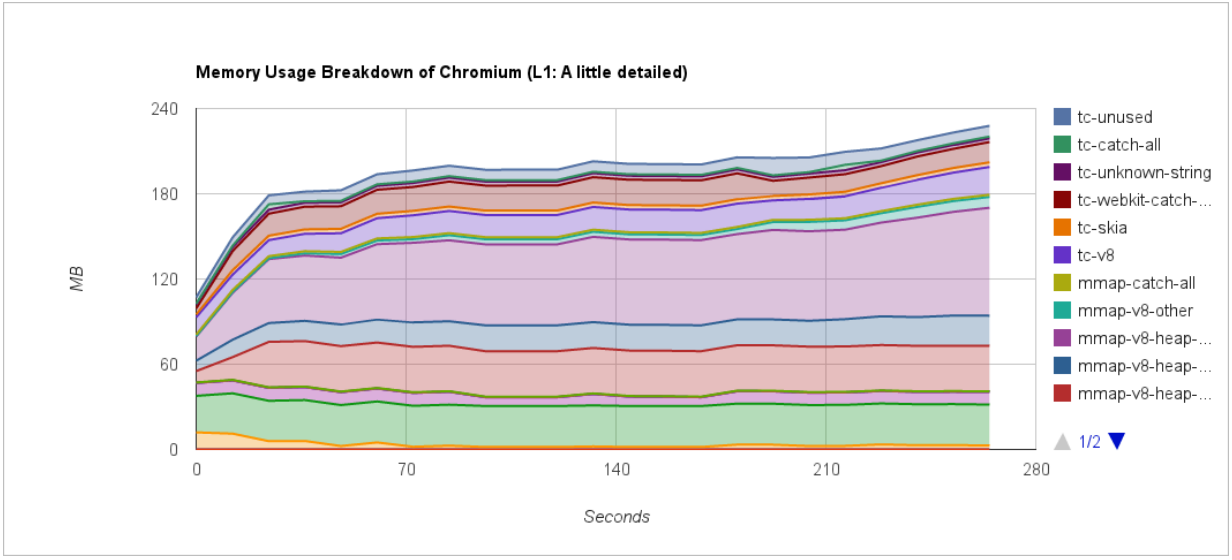
We start pattern matching from simple regular expression. For example,

```
.*WTF::StringImpl::createUninitialized WTF::String::fromUTF8.*
```

It matches `StringImpl` creation from UTF-8 strings. Advanced classification algorithms, like support vector machines, may be available in future.

Pattern matching doesn't work perfectly, of course. It's not accurate. But, we prefer rough and easy understanding to accurate profiling only for expertised engineers.

Graphs are naturally good to present profiling results. Stacked graphs are better since DMP classifies a total memory usage like following.



Related Work

Difference from TCMalloc's heap-profiler

DMP consists of

1. a dumper which dumps memory information, and
2. a post-mortem script which analyzes the dumped data.

They're similar to *heap-profiler.cc* (dumper) and *pprof* (post-mortem) in TCMalloc.

TCMalloc's dumper and DMP's dumper are different. DMP's dumper extends TCMalloc's dumper, and adds more information from operating system. For example,

- resident memory information from Linux /proc/.../pagemap, and
- correct mmap information separated from malloc.

More information is planned to be added. For example,

- timestamps of memory allocation,
- more information (e.g. swap) from /proc/.../pagemap,
- free'd memory chunks, not only their numbers, but also some status like "reused", "still unused but consuming memory", or "returned to OS",
- object types,
- and more...

For the post-mortem scripts, result graphs are completely different. The *pprof* script shows a network graph of stack traces for one snapshot. It requires expertised skills. DMP's script shows a stacked graph in a time-line for many snapshots.

Difference from Memory_Watcher

1) "Timeline" breakdown analysis and graphs

DMP's largest point is its "timeline" graphs like the graph above. Memory_Watcher snapshots only once. So, Memory_Watcher doesn't support such timeline analysis.

2) Automatic run on Linux

DMP runs semi-automatically on Linux. It's easy to prepare new bots. DMP is being integrated into endurance tests soon.

3) Additional information

DMP supports extended information like

- Available: physical page mapping and object type (by type profiler below)
- Planned: relation with JavaScript context, latest allocation timestamps, ...

Limitations, Extensions and Future Work

DMP is not aware of C++ type information and JavaScript objects information. They'd be so helpful to understand objects. In addition, we have a hypothesis that refptrs can hold a large cluster of objects, but DMP is not aware of them, too. We'll be extending DMP to solve these issues like following.

Identify types of C++ objects

It is done with a modified version of the Clang/LLVM compiler (as ASAN does). See the Design Doc.

Find refptrs which hold a large cluster of objects

Forgotten refptrs can hold a large cluster of objects. They may not be found by Valgrind if they are deallocated validly at a very end of the Chromium process.

Find relation with JavaScript objects

We are planning to do it by tracking C++ pointers and JavaScript references from some root objects through C++-JS bindings. It can be realized post-mortem with Linux core image files.

Besides them, in-vivo (runtime) analysis must be useful. I wonder if we could have in-vivo Deep Memory Profiling APIs for JavaScript which are enabled with a command line option --enable-memory-info.

Alternatives Considered

Adding instrumentation in object creation as written above. This approach would be fine since it enables detailed information about each object. But, it doesn't cover all memory allocation in the process.

Introducing a wrapper of malloc to append "caller tags" for each malloc call. It is a promising approach for tracking malloc with enough information. But it needs to replace calling statements everywhere. It's too hard...

Contact

- Dai Mikurube (dmikurube@chromium.org)